RNA Sekundärstrukturvorhersage (Studienarbeit)

Sebastian Bauer

26. Mai 2004

Inhaltsverzeichnis

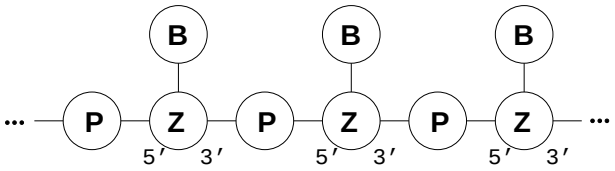


Abbildung 1: Schematischer Aufbau der RNA.

1 Einführung

Die genetische Information eines Organismus ist in den meisten Fällen in der DNA gespeichert. Um diese Information nutzen zu können, muss sie in eine andere Form, die Proteine, übersetzt werden, die dann als Enzyme oder Botenstoffe fungieren. Als Zwischenstufe dieser Übersetzung dient die RNA. Die Verarbeitung der genetischen Information in Proteine wird *Genexpression* genannt.

Es existieren auch Organismen, bei denen die RNA anstelle der DNA als permanenter Informationsspeicher verwendetet wird. Beispiele hierfür sind die RNA-Viren, zu denen auch das HIV zählt.

In neuster Zeit verdeutlichen sich die Anzeichen, dass die RNA im Organismus eine wesentliche größere Vielfalt von Funktionen zukommt, als bisher angenommen. So hat man entdeckt, dass es „Gene“[1] gibt, die ihre Wirkung nicht über Proteine, sondern komplett über RNA entfalten. Daher nimmt nun die Erforschung der RNA und ihre Funktionen für die Molekularbiologie einen wesentlich höheren Stellenwert ein.

Ähnlich wie bei Proteinen, bestimmt die räumliche Struktur solcher RNA ihre Wirkungsweise. Es sind Algorithmen der Bioinformatik, die die Forscher dabei unterstützen, diese Struktur vorherzusagen, aus der sie dann auf die Funktion schließen können. Diese Studienarbeitet widmet sich diesem Thema.

Wollen wir uns zuerst in diesem Einführungsabschnitt ein wenig mit den Grundlagen vertraut machen.

1.1 Aufbau und chemische Struktur der RNA

Chemisch betrachtet ist die RNA eine Kette aus vielen *Nukleoitden*. Deshalb spricht man auch häufig von einem *Polynukleotid*. Jedes dieser Nukleoitide besteht bei der RNA aus einem Ribosemolekül (d.h. einem Zucker mit 5 Kohlenstoffatomen), einem Phosphatrest und einer organischen Base. In der RNA kommen als organische Base Adenin, Guanin, Cytosin und Uracil in Frage. Dabei heißt Adenin *komplementär* zu Uracil und Guanin zu Cytosin. Bis auf Uracil entspricht das den Basen, die man auch in der DNA finden kann. In der DNA ist es durch das stabilere Thymin ersetzt.

Der Phosphatrest dient als Verbindung zwischen den Zuckermolekülen zweier benachbarter Nukleotide. Es verbindet das 3’ Kohlenstoffatom des Ribosemolekül eines Nukleotide mit dem 5’ Kohlenstoffatom des Ribosemolekül des anderen Nukleotides.

[1] sprechen Biologen von Genen, so meinen sie eigentlich die proteinkodierenden Abschnitte der DNA

Dieses Grundgerüst wird auch *Backbone* der RNA genannt. Die Basen befinden sich am Ribosemolekül.

Jeweils drei in der Kette nebeneinander liegende Nukleotide bilden ein Codewort (Codon), mit dessen Hilfe sich eine spezifische Aminosäure, die in ein Eiweiß (Protein) eingebaut werden soll, eindeutig bestimmen lässt.

Wie oben schon angedeutet, haben aber nicht nur Proteine im Organismus eine katalytische Funktion, sondern auch die RNA selbst. Die menschliche Erbsubstanz umfasst lediglich 2% proteinkodierende DNA. Während man anfangs annahm, dass der Rest meistens evolutionäre Überbleibsel sind, so hat man heute nachgewiesen, dass es in diesen Teilen der DNA ebenfalls funktional aktive Stellen gibt, die eben durch RNA ausgelöst werden.

1.2 Sequenz und Primärstruktur

Unter der *Sequenz* einer RNA verstehen wir eine Zeichenkette, die diese beschreibt. Genauer definiert sie die lineare Abfolge ihrer Basen und bestimmt damit im wesentlichen die physikalisch und chemischen Eigenschaften der RNA. Man nennt die Kette auch *Primärstruktur* der RNA.

Definition 1. Sei $\Sigma_{\text{RNA}} = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{U}\}$ (entsprechend den Anfangsbuchstaben der vier möglichen Basen). Eine RNA *Sequenz* ist dann eine Zeichenkette $r = r_0 r_1 \dots r_{n-1}$ über diesem Alphabet, d.h. $r \in \Sigma^*_{\text{RNA}}$. Wollen wir eine Teilsequenz zwischen i und j von r benutzen, schreiben wir $r_{i,j}$.

Beispiel 1. $r =$GCUAGGUCUAAGGUAGAAGAGUGCGAACUCGAGGGCAUGAAAGUGCGGUUACCACGAGGCGGCAUGUU

1.3 Sekundärstruktur

In der Natur tritt die RNA sehr selten als einzelne Kette auf. Vielmehr können sich Paarungen zwischen einzelnen Basen, die sich auf ein und dem selben RNA Strang befinden, ausbilden. Man sagt dazu, dass ein RNA Strang auf sich selbst *faltet*. Hauptsächlich entstehen Paarungen zwischen komplementäre Basen, aber auch zwischen Guanin und Uracil[2] und viel seltener zwischen anderen Basen. Nach welchen Prinzipien dies erfolgen kann, erörtern wir in einem späteren Abschnitt. Die *Sekundärstruktur* einer RNA beschreibt nun welche Basen mit welchen gepaart sind. Sie enthält dadurch mehr Informationen als die Sequenz.

Sei $[n]$ die Bezeichnung für die endliche Menge $\{0, \dots, n-1\} \subset \mathbb{N}$.

Definition 2. Eine *Sekundärstruktur* ist ein 2-Tupel $S = (r, P)$ mit r als Sequenz der RNA und der Menge $P \subset [|r|]^2$ von Paaren (i, j) mit $i < j - 1$[3], so dass für $(i, j), (i', j') \in P$ und $i = i'$ gilt: $j = j'$. Für eine Sekundärstruktur einer Teilsequenz zwischen i und j schreiben wir $S_{i,j}$.

[2] Man nennt diese drei Arten der Basenpaarungen auch *kanonisch*

[3] da die Natur keine zu scharfe Biegungen in der RNA zulässt gilt hier sogar, $i < j + 4$

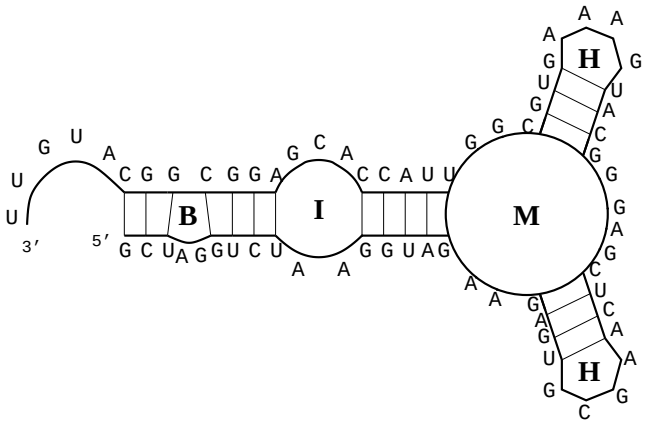


Abbildung 2: Mögliche Sekundärstruktur unserer Beispiel RNA

Abbildung 2 zeigt eine Darstellungsform der Sekundärstruktur auf. In dieser können wir verschiedenartige Anordnungen beobachten. Wir erkennen zunächst so genannte *Stems*. Dies sind zusammenhängende Strukturen, die nur aus Basenpaarungen bestehen. Alle weiteren Anordnungen bezeichnen wir als *Schleifen*. Dabei unterscheiden wir zwischen *Hairpin Loops*, Ausbuchtungen (*Bulge*) und den *Interior Loops* bzw. den *Multi Loops*.

Die Sekundärstruktur lässt sich auch sehr gut innerhalb eines Graphen darstellen, wobei ein Knoten einem Nukleotid entspricht. Eine Kante zwischen zwei Knoten verläuft, wenn sie Nachbarn in der Sequenz sind (die Menge aller solcher Kanten entspricht dem Backbone) oder wenn sie eine Paarung aufweisen. In Abbildung 3 ist ein solcher Graph dargestellt.

Eine weitere Möglichkeit die Sekundärstruktur aufzuzeigen ist die Verwendung eines *Klammerausdrucks*. Die Information über die Paarungen wird hier in einer zusätzlichen Zeichenkette $p = p_0 p_1 \dots p_{n-1}$ festgehalten. Ein Zeichen in p_i gibt an, ob die Base an der Position i gepaart ist. Für eine ungepaare Base schreiben wir ein ':'. Für gepaarte Basen schreiben wir an den Positionen öffnende bzw. geschlossene Klammern, so dass ein korrekter Klammerausdruck entsteht.

Beispiel 2.
$r =$ GCUAGGUCUAAGGUAGAAGAGUGCGAACUCGAGGGCAUGAAAGUGCGGUUACCACGAGGCGGCAUGUU
$p =$ (((::((((::(((((::((((::::)))):::::((((::::)))))::))))):::)))))):::::

1.4 Pseudoknoten

Mitunter kann es vorkommen, dass sich eine Base einer Schleife (die also noch nicht gebunden ist), mit einer Base außerhalb dieser Schleife paart. Abbildung 4 macht einen solchen Fall deutlich.

Definition 3. Sei $S = (r, P)$ eine Sekundärstruktur zu einer RNA. Eine RNA enthält *Pseudoknoten*, wenn für mindestens zwei Paare $(i, j), (k, l) \in P$ gilt: $i < k < j < l$.

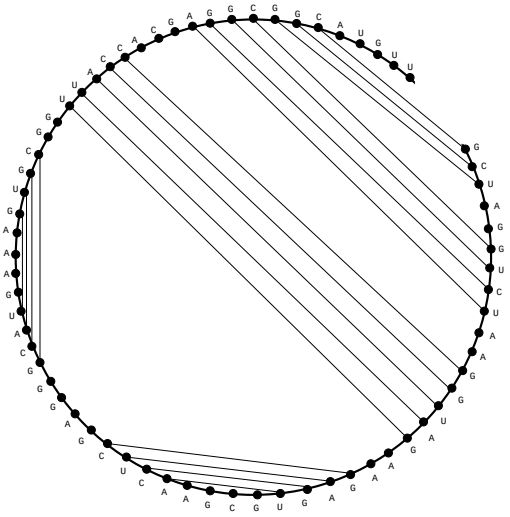


Abbildung 3: Graphendarstellung der Sekundärstruktur. Die Backbone ist der übersichtshalber kreisförmig dargestellt.

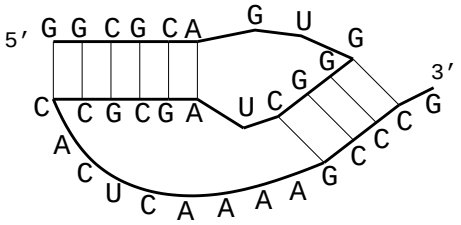


Abbildung 4: Sekundärstruktur einer RNA mit Pseudoknoten.

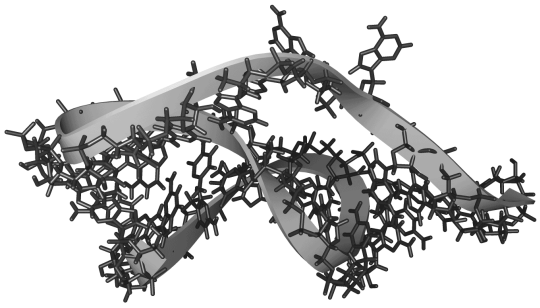


Abbildung 5: Die zur RNA aus Abbildung 4 gehörige Tertiärstruktur.

Ordnet man in der Graphendarstellung einer solchen RNA die Backbone so an wie in Abbildung 3, erhalten wir sich schneidende Kanten. In der Klammerndarstellung kennzeichnet man dies, in dem man für die Stems verschiedenartige Klammern verwendet. Die in dieser Studienarbeit betrachteten Strukturen lassen sich alle mit zwei verschiedenen Klammerpaaren darstellen.

Beispiel 3.
r =GGCGCAGUGGGCUAGCGCCACUCAAAAGCCCG
p =((((((::[[[[:))))))::::::::]]]]:

1.5 Tertiärstruktur

Unter der *Tertiärstruktur* verstehen wir die räumliche Anordnung der RNA. In Abbildung 5 sehen wir eine RNA in dieser Weise dargestellt. Was man in der Sekundärstruktur nicht direkt ablesen kann, ist, dass im Raum die *Stems* eine so genannte α-*helikane* Struktur ausbilden. Diese finden wir auch in der DNA vor. Das charakteristische Merkmal daran ist, dass sich die einzelnen Basenpaare um eine imaginäre gemeinsame Achse winden. Dies wirkt sich insgesamt stabilisierend auf die Struktur auf.

2 Vorhersage der RNA Sekundärstruktur

Während die Sequenz zwar die Funktion einer RNA bestimmt, kann man ihre Funktion und Eigenschaften nicht direkt aus ihr folgern. Erst mit Hilfe der Sekundärstruktur oder gar der Tertiärstruktur ist dies möglich. Es ist daher von besonderem Interesse, diese höheren Ordnungen möglichst automatisiert, ausgehend von den Sequenzen zu ermitteln. Wir wollen uns hier ausschließlich mit der Vorhersage der RNA Sekundärstruktur beschäftigen.

Der Grundgedanke, wie man eine solche Vorhersage treffen kann, basiert auf dem Fakt, dass die Moleküle natürlicherweise energetisch günstige Verbindungen eingehen. Die naive Idee wäre nun, dass wir alle möglichen Sekundärstrukturen aufzählen und diese

Basenpartner	**freie Energie (in $\frac{\text{kcal}}{\text{mol}}$)**
G–C	-3
A–U	-2
G–U	-1
alle Anderen	0

Tabelle 1: Durch Messungen ermittelte Werte zur freien Energie bei 37°C, siehe [3]

dann energetisch bewerten. Die Struktur mit günstigster Energie ist unsere Vorhersage. Die Anzahl der möglichen Sekundärstrukturen ist jedoch exponientiell zur Länge der RNA Sequenz, so gibt es z.B. schon für Sequenzen der Länge 200 mindestens 10^{50} mögliche Strukturen. Für größere Sequenzen führt diese Ansatz also nicht zu einer schnellen Lösung. Es wird daher nach anderen Wegen gesucht.

Wir wollen hier das Wesentliche eines sehr etablierten Algorithmus vorstellen. Dieser ist in der in der Lage, eine die Sekundärstruktur einer RNA vorherzusagen, wobei er jedoch keine Pseudoknoten in Betracht zieht. Ihm folgt noch ein kurzer Überblick über einen anderen Ansatz.

2.1 Vorhersage mittels Dynamischer Programmierung

Der hier gezeigt Algorithmus entspricht dem in [3] behandleten Verfahren und basiert auf dem Prinzip der Dynamischen Programmierung. Es werden die Werte aus der freien Energie von Teilstrukturen bzw. von Basenpaarungen abgeleitet. Werte für die freie Energie von Basenpaaren, helikalen Strukturen und Schleifen basieren auf experimentellen Messungen. Aus diesen Messungen und weiteren thermodynamischen Überlegungen können so Regelwerke für die freien Energien entwickelt werden. Der Versuch, diese Regeln den tatsächlichen Verhältnissen anzupassen, macht die entsprechenden Algorithmen sehr kompliziert. Daher werden wir uns nur auf ein sehr einfaches Regelwerk beschränken, und nehmen vorerst an, dass die Struktur allein aus den Energiebeiträgen von Basenpaarungen abgeleitet werden kann.

Sei $\alpha(a, b)$ mit $a, b \in \Sigma_{\text{RNA}}$ eine Funktion, die uns die freie Energie zwischen bei Paarbildung von einer Base a mit b liefert. Hierfür zeigt Tabelle 1 realistische Funktionswerte auf.

Nun sei $S = (r, P)$ eine wie oben definierte Sekundärstruktur. Für die freie Energie der Sekundärstruktur schreiben wir $E(S)$. Sie ergibt sich laut unserer Annahme lediglich aus der Summe der freien Energien der Paarungen, d.h:

$$E(S) = \sum_{(i,j)\in P} \alpha(s_i, s_j) \tag{1}$$

Wir suchen nun so ein S für welches $E(S)$ minimal ist. Für die freie Energie einer Teilsekundärstruktur zwischen i und j schreiben wir $E(S_{i,j})$.

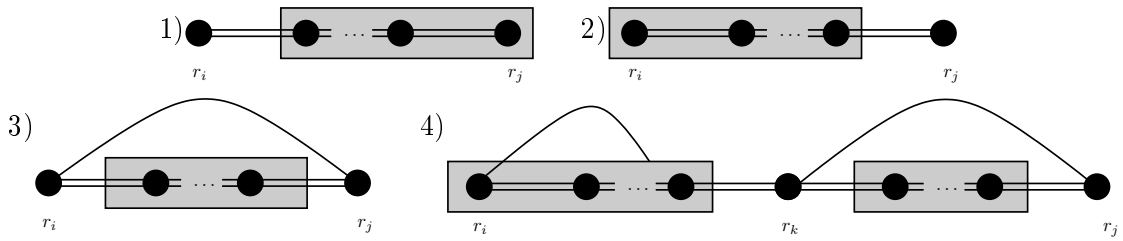


Abbildung 6: 4 Fälle, die bei unserem Modell eintreten können

Wir wollen nun eine Rechenvorschrift für $E(S_{i,j})$ bestimmen. Da die Natur keine starken Biegungen zulässt, haben wir bereits eine erste Regel gefunden:

$$E(S_{i,j}) = 0 \text{ für } j - i < 4$$

Es existieren nun vier weitere Fälle, die eintreten können (siehe Abbildung 6):

1. Die Base r_i geht mit keiner Base aus $r_{i+1,j}$ eine Verbidnung ein. Die Energie entspricht dann der Energie der Teilstruktur $S_{i+1,j}$.

2. die Base r_j geht mit keiner Base aus $r_{i,j-1}$ eine Verbidnung ein. Die Energie entspricht dann der Energie der Teilstruktur $S_{i,j-1}$.

3. die Base r_i geht mit r_j eine Verbindung ein. Die Energie setzt sich aus $\alpha(r_i, r_j)$ und der Energie für die Teilstruktur $S_{i+1,j-1}$ zusammen.

4. die Basen r_i und r_j sind Elemente unterschiedlicher Basenpaarungen. Dann geht die Base r_j mit einer Base r_k für $i < k < j$ eine Bindung ein. Die Energie setzt sich aus den Teilstrukturen $S_{i,k-1}$ und $S_{k,j}$ zusammen, wobei das k gewählt wird, für das die Summe minimal ist.

Da wir eine möglichst kleines $E(i, j)$ finden wollen, müssen wir alle Fälle nacheinander durchprobieren und uns für den besten entscheiden. Wir kommen zu folgendem Ergebnis:

$$E(S_{i,j}) = \min \begin{cases} E(S_{i+1,j}) & \text{(Fall 1)} \\ E(S_{i,j-1}) & \text{(Fall 2)} \\ \alpha(r_i, r_j) + E(S_{i+1,j-1}) & \text{(Fall 3)} \\ min_{i<k<j}\{E(S_{i,k-1}) + E(S_{k,j})\} & \text{(Fall 4)} \end{cases}$$

Wir haben somit eine rekursive Rechenvorschritt gefunden, die für Dynamische Programmierung geeignet ist. Fall 4 hat eine Laufzeit von $O(n)$ während alle anderen Fälle in konstanter Zeit abgearbeitet werden können. Folglich ergibt sich, wegen der Iteration über zwei Variablen, um die n^2 Einträge der Tabelle zu füllen, eine Komplexität von $O(n^3)$. Merkt man sich für jeden Strukturabschnitt, welches Fall man gewählt hat, so lässt sich am Ende der Berechnung die Sekundärstruktur durch Zurückverfolgung ableiten.

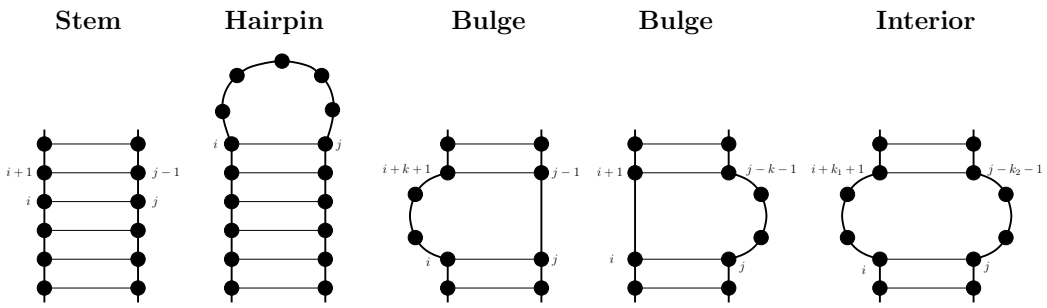


Abbildung 7: Anordnungen von RNA Strukturen

Die Annahme, dass die Struktur nur aus den Energiebeiträgen von einzelnen Basenpaarungen abgeleitet werden kann, ist nicht realistisch. So wirken sich im Raum helikale Strukturen (also viele aufeinanderfolgende Basenpaare) insgesamt stabilisierend aus, während Schleifen je nach Art verschieden schwer destabilisieren. Um diese Tatsache zu modellieren, verallgemeinert man Fall 3, indem die möglichen Anordnungen (siehe Abbildung 7) mit in die Energiebewertung einfliessen. Hierfür definieren wir neue Funktionen:

$$\begin{aligned}
\beta(k) &= \text{Energieterm eines } \textit{Bulges} \text{ der Größe } k \text{ (destabilisierend)} \\
\gamma(k) &= \text{Energieterm eines } \textit{Interior Loops} \text{ der Größe } k \text{ (destabilisierend)} \\
\xi(k) &= \text{Energieterm eines } \textit{Hairpin Loops} \text{ der Größe } k \text{ (destabilisierend)} \\
\eta &= \text{Energieterm für } \textit{Stacking} \text{ Intereraktion (stabilisierend)}
\end{aligned}$$

Im Anschluss lautet unserer neuer Fall 3 wie folgt:

$$\min \begin{cases}
\alpha(r_i, r_j) + \eta + E(S_{i+1,j-1}) & (\textit{Stem}) \\
\alpha(r_i, r_j) + \xi(j-i-1) & (\textit{Hairpin Loop}) \\
\min\{\alpha(r_i, r_j) + \beta(k) + E(S_{i+1+k,j-1}), k \geq 1\} & (\textit{Bulge} \text{ von } i \text{ bis } i+k) \\
\min\{\alpha(r_i, r_j) + \beta(k) + E(S_{i+1,j-1-k}), k \geq 1\} & (\textit{Bulge} \text{ von } j-k \text{ bis } j) \\
\min\{\alpha(r_i, r_j) + \gamma(k_1+k_2) + E(S_{i+1+k_1,j-1-k_2}), k_1, k_2 \geq 1\} & (\textit{Interior Loop})
\end{cases}$$

Aufgrund des *Interior Loop* Terms erhöht sich die Komplexität wegen der zwei freien Parameter k_1 und k_2 zunächst auf $O(n^4)$. Es gilt jedoch

$$2 \leq k_1 + k_2 < j - i - 2 - l_{\min} = m_{\max} \leq n,$$

wobei $l_{\min}$ die kleinste erlaubte Loopgröße angibt. Speichert man nun vor Abarbeitung des Algorithmus die Energiewerte aller Loopgrößen, die für ein Paar (i, j) in Frage kommen können in einen Vektor $V_{i,j}$ (mindestens in $O(n^3)$ möglich), so ergibt sich:

$$\min\{\alpha(r_i, r_j) + \gamma(m) + V_{i,j}[m], 2 \leq m < m_{\max}\} \quad (\textit{Interior Loop})$$

und damit wieder eine Komplexität von $O(n^3)$.

Es gibt noch viele verschiedene Regelwerke, die ebenfalls in diesen Algorithmus eingebettet werden könnnen, um ein möglichst realtisches Modell zu erhalten. Mehr Information hierfür findet man in [2]. Diese erweiterten Regelwerke führen dazu, dass ca. 50-60% der Basenpaarung korrekt vorhergesagt werden könnnen.

2.2 Andere Ansätze

Neben den Algorithmen basierend auf Dynamischer Programmierung, existieren noch einige Methoden zur 2D-Strukturvorhersage, die auf anderen Konzepten aufbauen. Das Programmpaket STAR[4] enthält zum Beispiel einen Ansatz, der die Struktur mittels eines genetischen Algorithmus bestimmt [4]. Während die Erkennungsrate dieses Algorithmus mit 45% hier niedriger ist, als die des obigen Algorithmus, ist dieser in der Lage ebenfalls Pseudoknoten vorherzusagen.

3 Pseudoknoten mittels Dynamischer Programmierung

Diese Studienarbeit versucht herauszufinden, ob ein neuer Ansatz, der verspricht mittels Dynamischer Programmierung ebenfalls Pseudoknoten zu erkennen, momentan existierende Strukturen theoretisch vorhersagen kann. Dafür wurde ein Programm geschrieben. In diesem Abschnitt werden daher die zum Verständnis des Programmes benötigten Konzepte erläutert. Mehr Informationen hierzu findet man in [1].

3.1 Strukturen

Ein wichtiger Punkt in diesem Ansatz ist, dass die eigentliche Struktur des RNA Moleküls von seiner Sequenz getrennt wird.

Definition 4. Eine *0-Struktur* ist ein 2-Tupel $\sigma = (n, P)$ mit $n \geq 1$ und der Menge $P \subset [n]^2$ von Paaren (i, j) mit $i < j - 1$, so dass für $(i, j), (i', j') \in P$ und $i = i'$ gilt: $j = j'$.

Eine *1-Struktur* ist ein 3-Tupel $\sigma = (n, P, k)$ mit $n > 1$ und $1 \leq k < n$ und der Menge $P \subset [n]^2$ von Paaren (i, j) mit $i < j - 1$ oder $(i, j) = (k, k + 1)$, so dass für $(i, j), (i', j') \in P$ und $i = i'$ gilt: $j = j'$.

Wenn wir allgemein von Elementen der Strukturen sprechen, so meinen wir die Elemente aus $[n]$. Wir sagen, dass ein Element $i \in [n]$ *gepaart* ist, genau dann, wenn P die Tupel (i, j) oder (j, i) enthält.

Eine 0-Struktur ist also eine gewöhnliche Sekundärstruktur eines gefalteten RNA Strangs, jedoch ohne die eigentliche Sequenz. Eine 1-Struktur repräsentiert die Struktur zweier gefalteter mit einander verbundener RNA Stränge. D.h. es existiert eine Lücke zwischen dem $(k - 1)$-ten und k-ten Element. Da für 1-Strukturen $1 \leq k < n$ gilt, können wir $\sigma = (n, P, 0)$ bzw. $\sigma = (n, P, n)$ für 0-Strukturen benutzen. Spielt der Typ der Struktur keine Rolle, so können wir also $\sigma = (n, P, k)$ schreiben.

[4] http://wwwbio.leidenuniv.nl/ Batenburg/STAR.html

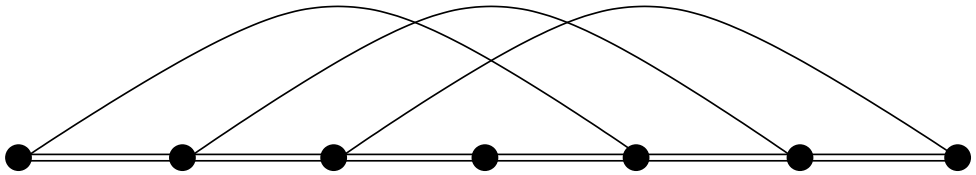


Abbildung 8: Struktur, die durch Klammerausdruck `([{:)]}` repräsentiert wird

Wie die Sekundärstruktur lassen sich 0- und 1-Strukturen günstig und intuitiv als Graphen darstellen. Ein solcher Graph sei definiert durch $G(\sigma)$. Er besteht aus n Knoten und besitzt die Kanten (i, j) falls $(i, j) \in P$. Weiterhin besitzt er die Kanten $(i-1, i)$ für $0 < i \leq n-1$ (entspricht der Backbone), wobei bei durch 1-Strukturen definierte Graphen die Kante $(k-1, k)$ fehlt.

Klammerausdrücke können ebenfalls verwendet werden und eignen sich zur Eingabe in Datenbanken. Die Lücke in den 1-Strukturen lässt sich durch ein Leerzeichen darstellen. Man muss allerdings beachten, dass nicht alle Strukturen sinnvoll durch Klammerausdrücke beschreibbar sind. Abbildung 8 zeigt eine Struktur, bei der schon 3 verschiedene Klammerpaare benötigt werden. Die Daten, die in dieser Studienarbeit untersucht worden sind, lassen sich jedoch alle mit 2 verschiedenen Klammerpaaren aufschreiben.

Wir benötigen nun noch eine Relation, mit der wir die Position der einzelenen ungepaarten und gepaarten Elementen in der Struktur festlegen und miteinander vergleichen könnnen.

Definition 5. Sei $\sigma = (n, P, k)$ eine Struktur. Die Relation $<_\sigma$ ist gegeben durch:

- $i <_\sigma j$, genau wenn $i < j$.
- $i <_\sigma (i', j')$, genau wenn $i < i' < j'$.
- $(i', j') <_\sigma i$, genau wenn $i' < i$.
- $(i, j) <_\sigma (i', j')$, genau wenn $i < i'$.

3.2 Kompositionen

Die Grundidee ist, dass größere Strukturen aus kleineren zusammengesetzt werden können. Wie das funktioniert, wollen wir hier nur in der intuitiven Variante – anhand der Graphen – beschreiben. Eine formale Definition findet man in [1].

Sei $\sigma = (n, P, k)$ eine Struktur. Es existieren zwei verschiedene Arten von Komposition. Die erste ist die Komposition entlang eines ungepaarten Elementes. Sei nun $\tau = (m, Q)$ eine 0-Struktur. Auf Strukturebene schreiben wir formal für die Kompostion entlang des Elementes i: $\sigma \circ_i \tau$. Die Operation ersetzt das Element i durch τ. In der Graphendarstellung würde dies bedeuten, dass wir Knoten i von $G(\sigma)$ durch $G(\tau)$ ersetzen.

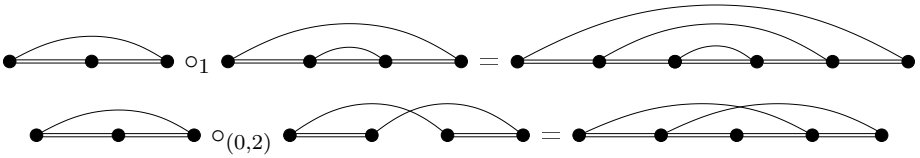


Abbildung 9: Kompositionen entlang eines Knoten und einer Paarung

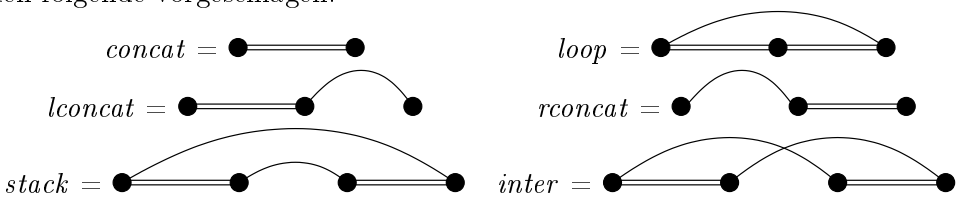
Die zweite ist die Komposition entlang gepaarter Elemente. Sei jetzt $\tau = (m, Q, l)$. Für die Operation entlang des Paares (i, j) schreiben wir $\sigma \circ_{(i,j)} \tau$. Sie entfernt das Paar (i, j) und fügt an dessen Stelle durch eine bestimmte Vorschrift τ ein. Graphisch wird die Kante (i, j) in $G(\sigma)$ durch $G(\tau)$ ersetzt und zwar so, dass der erste Strang von $G(\tau)$ den Knoten i und der zweite Strang den Knoten j ersetzt.

Beide Operationen ändern den Strukturtyp des ersten Arguments nicht. In Abbildung 9 finden wir zwei Beispiele zu den Operationen. Der i-te *Input* einer Struktur sei nun das i-te Element oder die i-te Paarung entsprechend der Ordnungsrelation $<_\sigma$ (beginnend bei 1). Weil die Komposition an einem Input andere Inputs nicht betrifft, können wir mehrere Kompositionen auf verschiedenen Eingaben „simultan“ ausführen. Wir schreiben hierfür $\sigma \circ (\tau_1, \ldots, \tau_o)$, wobei τ_i die Struktur für den i-ten Input und o den Input mit höchster Ordnung darstellen.

3.3 Generatoren

Wir wissen nun, wie wir aus kleineren Strukturen größere zusammensetzen können. Es macht Sinn die Anzahl der Strukturen, die zum Aufbau der großen Strukturen benötigt werden, festzulegen und möglichst klein zu halten. Dadurch wird die Suche nach einer passenden Struktur schneller. Wir nennen solche Strukturen *Generatoren* oder *Erzeuger*. Einelementige 0-Strukturen oder zweielementige 1-Strukturen schließen wir jedoch als Generatoren aus, da sie keine Strukturveränderung bewirken.

Für die Menge, die die zulässigen Generatoren beinhaltet, schreiben wir G. In [1] werden folgende vorgeschlagen:



Mit *concat*, *loop*, *lconcat*, *rconcat* und *stack* alleine können alle verschachtelten Strukturen (also ohne Pseudoknoten) erzeugt werden. Mit *inter* können zusätzliche Strukturen mit Pseudoknoten erzeugt werden. Wir werden später analysieren, wieviel biologisch relevante dies sind.

Generatoren verwenden wir immer als erstes Argument einer Komposition.

Beispiel 4.

$$\sigma = loop \circ ((4, \{(0,2),(1,3)\}, 2)_1, (2, \{\})_2)$$

Lässt sich eine Struktur $\sigma = (n, P, k)$ durch auf nacheinander angewandte Komposition auf Generatoren, die in der Menge G enthalten sind, erstellen, so sagen wir, dass die Struktur mittels der Generatoren aus G *erzeugbar* ist.

4 Programmdokumentation

Kommen wir jetzt zu der Beschreibung des Programmes, das vorhandene Sekundärstrukturen auf ihre *Erzeugbarkeit* hin untersucht. Nach einem kurzen Überblick über die Fähigkeiten, den Vorraussetzungen und der Installation des Programm wird hier hauptsächlich im 4. Unterabschnitt auf die Benutzerführung des Programmes eingegangen. Der letzte Unterabschnitt stellt einen kurzen Beispielablauf dar.

4.1 Merkmale

RNA Check besitzt folgende Merkmale

- unterstützt das Herunterladen von Datensätzen von der PseudoBase[5].
- eine graphische Benutzerumgebung
- die Generatorn sind dynamisch zur Laufzeit festlegbar

4.2 Voraussetzungen

Da das Programm in Java geschrieben ist, benötigt es eine vorher installierte Java Run Time Environment[6]. Es wurde mit der Version 1.4.2 getestet. Zusätzlich benutzt es den NekoHTML Parser, der wiederum auf Xerces vom Apache XML Projekt aufbaut. Als Oberflächen Toolkit wird das von der Entwicklungsumgebung Eclipse bekannte SWT benutzt.

4.3 Installation

Für Windows enthält das Programmarchiv bereits alle wichtigen Bibliotheken. Es genügt hier das Archiv einfach zu entpacken. Sollte ein anderes Betriebssystem verwendet werden, muss die entsprechne SWT Umgebung vorher noch eingerichtet werden.

[5] siehe http://wwwbio.leidenuniv.nl/ Batenburg/PKB.html. Sie beinhaltet wichtige Segmente von Sekundärstrukturen von RNA's mit Pseudoknoten.

[6] Aktuelle Versionen kann man über http://java.sun.com beziehen

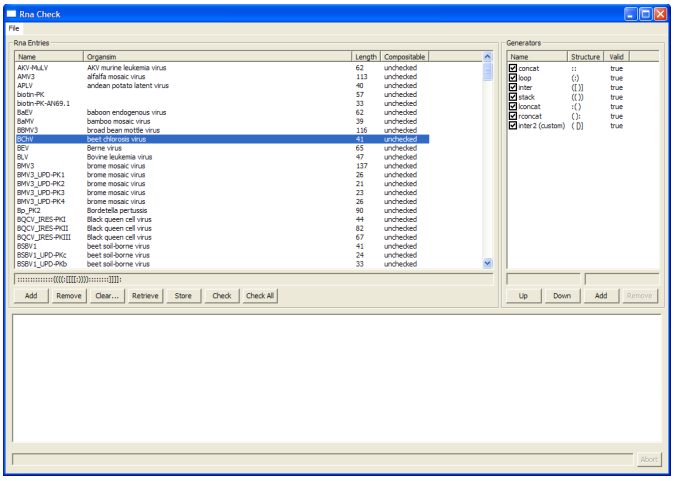


Abbildung 10: Screenshot des Programmfensters

4.4 Bedienung

Windows-Benutzer öffnen die *rnacheck.bat* Datei. Nach erfolgreichem Start erscheint das Hauptfenster. Es untergliedert sich im Wesentlichen in drei Bereiche. Im linken oberen Teil befinden sich die Verwaltung der RNA Einträge. Im kleineren rechten Bereich befinden sich die Generatoren. Der untere Teil dient zur Ausgabe von Logging und Statusmeldungen.

4.4.1 RNA

Alle verfügbaren RNA's werden in der Tabelle dargestellt. Die Tabelle besitzt Spalten für den Namen des Polynukleotides, den Organismus in dem es vorkommt und die Länge der Sequenz der RNA. Die letze Spalte gibt an, ob die Sekundärstruktur der RNA durch die Generatoren erzeugt werden kann, dabei heißt *unchecked*, dass sie noch nicht überprüft worden ist. Die Tabelle unterstützt Mehrfachselektion. Für die jeweils letzte selektierte RNA wird ihre Sekundärstruktur im Textfeld unterhalb der Tabelle gezeigt.

Momentan existieren zwei Möglichkeiten, die Tabelle zu füllen. Die einfachste ist, dass sich das Programm die Daten aus einer Datenbank wie der PseudoBase im Internet holt. Diesen Fall regelt der *Retrieve* Button. Da dies unter Umständen sehr lang dauern kann, lässt sich dieser Vorgang abbrechen. Im Statusfeld wird der Fortschritt der Operation angezeigt.

Die zweite ist das manuelle Eingeben der Daten. Mit dem *Add* Button werden neue RNA Einträge erzeugt, die wieder mit *Remove* entfernt werden können. Die Sekundärstruktur kann im Textfeld eingegeben werden, sofern der manuell erstellte Eintrag selektiert ist.

Bedienfeld	Funktion
Add	fügt eine neue Sekundärstruktur hinzu.
Remove	entfernt alle selektierten Einträge.
Clear	entfernt alle Einträge.
Retrieve	empfängt RNA Einträge aus unterstützten Datenbanken.
Store	macht die momentanen Einträge persistent.
Check	überprüft selektiete RNA Einträge auf Erzeugbarkeit.
Check all	überprüft alle RNA Einträge auf Erzeugbarkeit.
Texteingabefeld	Sekundärstruktur des selektierten Generators

Tabelle 2: Funktionen des RNA-Bereichs im Überblick

Bedienfeld	Funktion
Up	erhöht die Priorität des Generators.
Down	erniedrigt die Priorität des Generators.
Add	fügt einen neuen Generator hinzu.
Remove	entfernt den selektierten Generator.
linkes Texteingabefeld	definiert den Namen des Generators
rechtes Texteingabefeld	definiert die Struktur des Generators

Tabelle 3: Funktionen des Generatorbereichs im Überblick

4.4.2 Generatoren

Die Menge der Generatoren kann zur Laufzeit geändert werden. Verfügbare Generatoren werden in einer Tabelle aufgelistet. Ein Generator ist zunächst gekennzeichnet durch einen Namen und eine Struktur – entsprechende Spalten findet man in der Tabelle. Die Struktur wird über Klammerausdrücke dargestellt mit einem Leerzeichen für die Lücke bei 1-Struktur Generatoren. Es werden zwei verschiedene Klammerpaare unterstützt. Die letze Spalte gibt an, ob die Struktur des Generators korrekt ist. Zusätzlichen lassen sich gezielt Generatoren mittels der Checkbox an- und abschalten. Ein Generator dessen Checkbox angeschaltet ist, nennen wir *aktiv*.

Unterhalb der Tabelle lassen sich neue Generatoren erzeugen bzw. entferen. In den Textfeldern gibt man dann die gewünschten Daten ein. Die Priorität eines Generators ergibt sich aus der Position in der Tabelle. Je höher ein Generator gelistet ist, desto höher ist seine Priorität.

4.4.3 Überprüfen einer Sequenz

Das Überprüfen einer Struktur auf Erzeugbarkeit geschieht mittels *Check* Button im RNA Bereich, der alle selektierten Einträge darauf hin überprüft bzw. mittels *Check All* welches alle Einträge überprüft. Es werden lediglich die aktiven Generatoren benutzt, wobei Generatoren mit höherer Priorität bevorzugt werden (da es hier nur um Erzeug-

barkeit geht, spielt dies aber für das Resultat keine Rolle). Dieser Vorgang kann je nach Länge und Anzahl der Sequenzen einige Zeit in Anspruch nehmen, kann jedoch unterbrochen werden. Den aktuellen Fortschritt erkennt man im Statusfeld. Mögliche Fehler werden im Loggingfeld aufgeführt. Das Ergebnis der Operation, wird in der 4. Spalte der RNA Tabelle festgehalten.

4.5 Beispielablauf

Startet man das Programm zum ersten Mal, so ist die RNA Datenbank noch leer. Wir benutzen deshalb zunächst *Retrieve*, um die Tabelle zu füllen. Nachdem die Operation abgelaufen ist (die Tabelle dürfte dann einige 200 Einträge enthalten), benutzen wir *Store* und die Daten persistent zu machen.

Als nächstes wählen wir die RNA mit Namen „AKV-MuLV“ aus. Sie sollte ganz oben in der Tabelle zu finden sein. Die Sekundärstruktur dieser RNA sollte folgende Form besitzen:

```
::::::::::::(((((((((:[[[[[[[)))):))))::::::::::::::::::]]]]]]]:
```

Wir wollen jetzt überprüfen, ob diese Struktur durch die momentan aktiven Generatorn erzeugbar ist [7]. Da wir die RNA schon selektiert haben, genügt es auf den *Check*-Button zu klicken. Nach kurzer Zeit sollte in der *Compositable* Spalte ein *false* zu sehen sein, d.h. die Struktur ist nicht durch die Generatoren erzeugbar.

Deshalb fügen wir nun ein weiteren Generator der Struktur `( [)]` ein. Hierzu klicken wir auf *Add* im Generatorbereich, geben dem neuen Generator einen Namen und definieren seine Struktur jeweils in den einen der beiden Textfeldern. Zum Schluss aktivieren wir den Generator mittels der Chckbox.

Ein erneuter Klick auf *Check* sollte nun die *Compositable* Spalte auf *true* setzen. Erst durch diesen Generator ist die Struktur also erzeugbar.

5 Algorithmus

In diesem Abschnitt gehen wir auf den verwendeten Algorithmus ein, der testet, ob die Erzeugung einer gegebenen Struktur, durch vorgegebene Generatoren möglich ist. Wir betrachten in der Tat nur die Struktur, die eigentliche Sequenz interessiert uns hier nicht. Zunächst formulieren wir die Problemstellung genau.

> **Problemstellung** – Gegeben ist eine 0-Struktur $\sigma = (n, P)$ und eine Menge G von Generatoren. Lässt sich σ durch sukessives Komponieren der Generatoren aus G erzeugen?

[7] zu Programmstart sind das die in in [1] vorgeschlagenen

5.1 Schritt für Schritt

Eine erste Idee ist es, die Komposition mit Generatoren invers auszuführen. D.h. wir suchen unser σ elementweise nach der Struktur der Generatoren ab. Passt ein 0-Struktur Generator darauf, so ersetzen wir ihn durch ein ungepaartes Element. Passt ein 1-Struktur Generator darauf, so ersetzen wir seine beiden Strukturteile durch zwei miteinender verbundene Elemente. Die Struktur nach der Ersetzung ist dann unser neues σ. Wir wiederholen den Vorgang solange, bis kein Generator mehr passt oder bis σ nur noch aus ein ungepaarten Element besteht. Im letzen Fall bedeutet das, dass unser ursprüngliches σ durch die verwendeten Generatoren erzeugbar ist.

Diese Vorgehensweise lässt sich auf dem ersten Blick sehr schnell und einfach implementieren. Was passiert jedoch, wenn am Ende des Algorithmus σ mehr als ein Element beinhaltet? Nun, in diesem Fall sagt uns das Resultat, dass die Struktur nicht durch die Generatoren erzeugbar ist, die wir verwendet haben. Auf dem Weg zum Ergebnis ist es aber durch aus möglich, dass ein anderer Generator zum Erfolg geführt hätte. Das bedeutet, wir wissen zwar, dass bei Erfolg die Struktur erzeugbar ist, beim Fall eines Fehlschlag können wir jedoch nichts mit Sicherheit aussagen. Abhilfe schafft hier, dass im Falle eines Fehlschlags, zurückgegangen und dann einen anderen Generator versucht wird. Wir bekommen dann aber exponientelle Laufzeit, was den Algorithmus nur für sehr kleine Strukturen brauchbar macht.

5.2 Memoization

Wir wollen nun eine Lösung finden, die schneller zum Ziel führt. Hierzu nutzen wir aus, dass eine Struktur $\sigma = (n, P, k)$ erzeugbar ist, wenn man sie auf bestimmte Art und Weise in kleinere *Teilstrukturen* zerlegen kann, die entweder ihrerseits durch Generatoren erzeugbar sind oder eine *triviale* Struktur haben. Die triviale 0-Struktur behinhaltet genau ein Element, während die triviale 1-Struktur genau zwei mit einander gepaarte Elemente umfasst. Ein weiterer Fall, der sich schnell abhandlen lässt, sind Teilstrukturen, die gepaarte Elemente in anderen Teilstrukturen besitzen. Solche Teilstrukturen sind für sich genommen nicht erzeugbar. Übrig bleibt noch das Problem, nach welcher Art und Weise wir die ursprüngliche Struktur in Teilstrukturen zerlegen.

Sei σ wie oben definiert und $\gamma = (m, Q, l)$ ein Generator mit $m \leq n$. Zusätzlich gilt natürlich $m > 1$, falls γ 0-Struktur bzw. $m > 2$ falls γ 1-Struktur. Wir unterteilen nun das Interval $[0, n-1]$ in m Unterintervalle mit $I_i = [a_i, b_i]$, wobei $a_0 = 0$, $b_{m-1} = n$ und es weiterhin gilt: $a_{i+1} = b_i - 1$ für $0 \leq i < n$. Über die übrigen Parameter iterieren wir.

In jedem Iterationsschitt, können wir jetzt die Intervalle I_i jedem Element von γ eindeutig zuordnen. Das bedeutet für Inputs, die lediglich ein ungepaartes Element umfassen, haben wir genau ein Interval und für Inputs, die sich aus einem Paar zusammensetzen, haben wir zwei Intervalle. Falls nun je nach Art des Inputs alle Strukturen begrenzt durch die Intervalle erzeugbar sind (Rekursion), dann ist σ erzeugbar. Abbildung 11 sehen wir eine solche Zuordnung anschaulich.

Eine Teilstruktur kann gleichzeitig Teilstruktur von verschieden größeren Strukturen sein. Wenn wir jetzt die Ergebnisse der Iterationsschritte in Tabellen halten, so dass

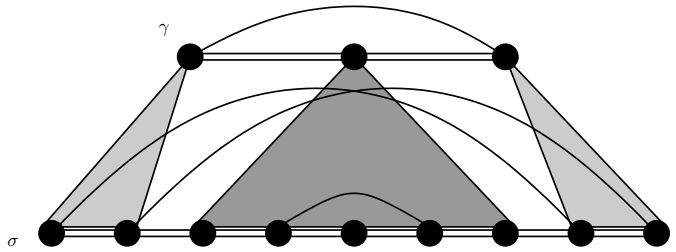


Abbildung 11: Zuordnung der Inputs von γ auf die Teilstrukturen von σ

nur einmal bestimmt wird, ob eine Teilstruktur erzeugbar ist oder nicht, können wir die Zeitkomplexität erheblich verbessern. Entsprechend der Strukturtypen haben wir dabei zwei Tabellen. Die Tabelle für 0-Strukturen hat als Spalten den Beginn und die Länge einer Teilstruktur, während die Tablle für 1-Strukturen zwei zusätzliche Spalten für den zweiten Strang der 1-(Teil)Struktur besitzt.

Es handelt sich hierbei um die klassische Dynamische Programmierung. Wir arbeiten Top Down und man spricht deswegen von der *Memoization* Variante. Während diese Variante dazu führt, dass im Falle einer Erzeugbarkeit nicht alle möglichen Teilstrukturen untersucht werden müssen, hat ihre Rekursivität den Nachteil, neben dem Platz für die Tabellen, zusätzlichen Platz für den Stack zu benötigen. Da wir bei jedem Schritt die Größe der zu untersuchenden Teilstruktur auf Grund der Festlegung der Intervalle um wenigstens 1 verringern, haben wir jedoch nicht mehr als n Aufrufe und man kann ihn mit $O(n)$ abschätzen. Unsere Zeitkomplexität liegt grob bei $O(n^{4+m})$, wobei m die Anzahl der Elemente eines größten Generators entspricht abzüglich eins und ergründet sich aus der Iteration über die Intervalle. Das Polynom n^4 resultiert aus der Tabelle für 1-Strukturen, die die Dimensionen $n \times n \times n \times n$ besitzt. Damit ergibt auch, dass der Gesamtplatzbedarf bei $O(n^4)$ liegt.

5.3 Mathematische Rechenvorschrifft

Wir können dieses Verfahren in einer Rekursionsgleichung mathematisch festhalten. Mit S bezeichnen wir die Menge aller Strukturen.

Für ein $\sigma = (n, P)$ und $i \leq j$ sei $\sigma_{i,j}$ die 0-Struktur, die aus σ entsteht und nur die Elemente von i bis einschließlich j enthält. Existieren in σ Paare (a, b), die $i \leq a, b \leq j$ nicht entsprechen, so ist $\sigma_{i,j} = \epsilon$. In diesem Fall exisitieren Paarungen von Elementen im Intervall zu Elementen außerhalb des Intervalls. Analog dazu ist für $\sigma_{i,j,k,l}$ mit $i \leq j < k \leq l$, die aus σ gewonnene 1-Struktur, deren erster Strang die Elemente i bis j und zweiter Strang die Elemente k bis l enthält, falls σ nur Paare (a, b) enthält mit $i \leq a, b \leq j$ oder $k \leq a, b \leq l$. Sonst gilt $\sigma_{i,j,k,l} = \epsilon$.

Nun sei C eine Funktion $S \to \mathbb{B}$ mit $\mathbb{B} = \{0, 1\}$, die berechnet, ob eine gegebene Struktur erzeugbar ist. Dabei gehen wir hier exemplarisch von einer Generatormenge bestehend aus *concat*, *loop*, *stack* und *inter* aus. C sieht wie folgt aus:

$$
\begin{aligned}
C(\epsilon) &= 0 \\
C(\sigma_{i,i}) &= 1 \\
C(\sigma_{i,i,j,j}) &= 1 \\
C(\sigma_{i,j}) &= \max \begin{cases} C(\sigma_{i,k-1}) \cdot C(\sigma_{k,j}) & i < k \leq j \quad (concat) \\ C(\sigma_{i,k-1,l,j}) \cdot C(\sigma_{k,l-1}) & i < k < l \leq j \quad (loop) \end{cases} \\
C(\sigma_{i,k,m,o}) &= \max \begin{cases} C(\sigma_{i,j-1,n,o}) \cdot C(\sigma_{j,k,m,n-1}) & i < j \leq k, m < n \leq o \quad (stack) \\ C(\sigma_{i,j-1,m,n-1}) \cdot C(\sigma_{j,k,n,o}) & i < j \leq k, m < n \leq o \quad (inter) \end{cases}
\end{aligned}
$$

An dieser Rekursionvorschrifft erkennt man ebenfalls, dass Dynamische Programmierung zur Lösung benutzt werden kann. Um die Erzeugbarkeit von $\sigma = (n, P)$ zu überprüfen, muss man $C(\sigma_{0,n-1})$ bestimmen.

6 Implementierung

Nachdem wir uns mit dem eigentlichen Algorithmus vertraut gemacht haben, kommen wir zur Implementation. Die Struktur des Programmes ist grob vier Teile untergliedert. Klassen, die verantwortlich für das Benutzerinterface sind befinden sich im *ui* Package. Klassen, die die RNA Datenbank verwalten sind im *database* Package zu finden, während Klassen für den eigentlichen Algorithmus in der *structure* Package eingegliedert sind. Hilfsklassen befinden sich im *util* Paket.

In den folgenden Unterabschitten wird jedes einzelne Paket beschrieben. Der Schwerpunkt liegt allerdings auf dem *structure* Paket von denen auch einige Codebeispiele aufgeführt werden. Für genauere Details innerhalb der anderen Pakete sei hier auf die ausführlich kommentierten Quellen verwiesen.

6.1 Package util

Dieses Paket enthält nur eine Klasse *Mutex*. Sie basiert auf den Quellen des *jsync* Pakets (http://www.garret.ru/ knizhnik/java.html) und wird verwendet, um kritische Abschnitte zu schützen.

6.2 Package ui

Die Benutzerumgebung wird in diesem Paket implementiert. Es beinhaltet den Einstiegspunkt des Programmes, das sich nach der nach einer Initalisierungsphase in einem Eventloop befindet, so lange bis der Benutzer die Beendung des Programmes wünscht.

- *RnaApplication* implementiert den Einstiegspunkt des Programmes (d.h. die *main* Methode) initialisiert die Umgebung und führt den Eventloop aus.

- *RnaShell* implementiert das Fenster und alle enthaltenes Bedienelemente des Programmes.
- *GeneratorGroup* implementiert den Generator-Bereich.
- *RnaDatabaseGroup* implementiert den kompletten Datenbank-Bereich.
- *RnaMenu* implementiert die Menüleiste.
- *UserLogger* ist eine statische Klasse und leitet die Logging- und Statusmeldungen von anderen Paketen zum Benutzerinterface.

6.3 Package database

Es existieren im Moment 3 Klassen in diesem Paket:

- ein Objekt der Klasse *RnaEntry* repräsentiert einen kompletten RNA Eintrag bestehend aus Namen, Organismus, Sequenz und der Sekundärstruktur des Polynukleotides. Zusätzlich Attribute sind vorgesehen, um anzugeben, ob die Sekundärstruktur erzeugbar ist, oder nicht.
- *RnaDatabase* ist eine statische Klasse. Sie stellt die Schnittstelle zwischen verschieden Datenbanken und anderen Bereichen des Programmes dar.
- *PseudoBase* ist ebenfalls eine statische Klasse, die für die Anbindung zur Pseudoknoten Database zuständig ist und die die ermittelten Daten zur RnaDatabase weiterleitet.

  Die PseudoDatabase beinhaltet Einträge von bekannten RNA's mit Pseudoknoten. Da es im Prinzip gewöhnliche Internetseiten sind, kann man die Informationen nur mittels HTTP Protokol und anschliessendes Parsing gewinnen. Der Aufbau der Datenbank ist so, dass es eine Seite gibt, die alle Einträge auflistet und für jeden Eintrag ein Querverweis beinhaltet, der zu mehr Informationen führt.

  Die Methode *receive()* läd nun über eine HTTP Verbindung diese HTML Datei herunter. Die enthaltenen Links werden mittels NekoHTML für jeden Eintrag extrahiert und weiterverfolgt, um daraus die Sekundärstruktur zu erhalten. Da all das ein sehr langer Prozess ist, wird diese Aufgabe in einem extra Thread erledigt, dem Hauptprogramm ist es also so weiterhin möglich auf Benutzerinteraktionen zu reagieren.

6.4 Package structure

Dieses Paket enthält hauptsächlich folgende Klassen:

- Die Klasse *Composer* nimmt eine Struktur auf und testet, ob eine Komposition mit gegebenen Generatoren möglich ist.

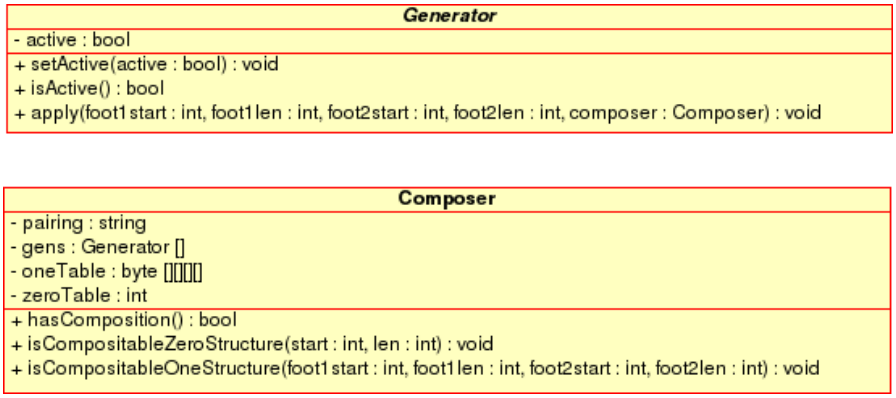


Abbildung 12: UML Diagramm einiger wichtiger Klassen im *structure* Paket

- Die abstrakte Klasse *Generator*. Sie definiert die Methoden, die Generatoren besitzen müssen. Sie ist Basisklasse aller spezifischen Generator Klassen wie z.B. *GeneratorConcat*, *GeneratorLoop* und *GeneratorASCII*.
- Die statische Klasse *Generators*. Sie verwaltet die Generatoren und stellt die dafür notwendigen Methoden bereit.
- *Structure* repräsentiert eine RNA Struktur innerhalb des *structure* Pakets und stellt ein Wrapper für *Composer* dar.

Daneben existieren noch einige historische bedingte Hilfsklassen.

- Ein Objekt der Klasse *Nucleotide* repräsentiert ein Nucloitid der RNA, dass ein Vorgänger, Nachfolger und ein gepaartes Nukleotid haben kann.
- *NucleotideList* repräsentiert die gesamte RNA Struktur, wobei diese nicht durch eine Zeichkette gehalten wird, sondern eine der dopplet verketteten Liste ähnliche Datenstruktur.
- Die abstrakte Klasse *GeneratorHandle*

Die Klassen werden im Moment benutzt, um eine Sekundärstruktur auf Korrektheit zu überprüfen (d.h. korrekte Klammerung) und um die Struktur des Generatoren für die Klasse *GeneratorASCII* zu ermitteln. In Zukunft könnten sie allerdings verwendet werden, um eine lange RNA Struktur vor der eigentlichen Existenzprüfung einer Komposition zu kürzen.

Gehen wir jetzt noch etwas genauer auf ausgewählte Klassen ein. Eine Überblick welche Methoden und Attribute diese Klassen haben finden wir Abbildung 12 in Form eines UML Diagrammes.

Der grobe Ablauf zur Überprüfung ist folgender: Zuerst wird die Methode *isCompositableZeroStructure()* der Klasse *Composer* aufgerufen (entspricht der im vorherigen

Kapitel eingeführten Funktion C), die dann die einzelnen *apply()* Methoden der Generatoren benutzt, um zu entscheiden, ob die gegebene Unterstruktur erzeugbar ist. Innerhalb der *apply()* Methoden werden wiederum die *isCompositableZeroStructure()* bzw. *isCompositableOneStructure()* mit kleineren Strukturen aufrufen, solange bis der triviale Fall erreicht ist. Alle Ergebnisse werden in Tabellen festgehalten, um sicher zu stellen, dass jede Unterstruktur nur einmal überprüft wird.

6.4.1 Die Klasse Composer

Die Klasse besitzt vier Attribute. Sie werden beim Anlegen eines Objektes gesetzt. Es ist nicht vorgesehen, eine spätere Änderung an ihnen durchführen zu können.

- *pairing* beschreibt die zu untersuchende Struktur als Klammerausdruck.
- *gens* ist ein Feld von *Generator* Instanzen. Es beinhaltet alle Generatoren, die beim Überprüfen auf Komposition verwendet werden dürfen.
- *zeroTable* und *oneTable* stellen unsere Tabellen für alle möglichen Teilstrukturen dar, also die Tabellen, in denen abgelegt, ob diese Teilstrukturen erzeugbar ist oder nicht. Insbesondere bedeutet hier ein Wert von 0, dass sie Teilstruktur, von nicht überprüft wurde und -1, dass die Teilstruktur mit den Generatoren nicht erzeugbar ist. Alle anderen Werte, heissen, dass die Teilstruktur erzeugbar ist.

Die Methode *isCompositableZeroStructure* untersucht, ob eine Strukturabschitt, definiert durch Startpunkt und Länge (Parameter *start* und *len*), als eine 0-Struktur erzeugbar ist. Gehen wir nun Schritt für Schritt durch den Code dieser Methode.

Im Einstiegspunkt der Methode wird anhand der *zeroTable* überprüft, ob bereits ein Ergebnis für diesen Strukrurabschnitt vorliegt. Ist dies der Fall so geben wir je nach Erzeugbarkeit der Struktur einen entsprechenden Rückgabewert zurück:

```
if (zeroTable[start][len] == FAILURE_INDEX) return false;
if (zeroTable[start][len] != UNCHECKED_INDEX) return true;
```

Als nächstes wird der ausgewählte Ausschnitt auf Korrektheit überprüft. Die Klasse *NucleotideList* stellt die notwendigen Hilfsmittel bereit – sie wirft eine *IllegalArgumentException* sollte der Ausschnitt nicht korrekt sein. Das ist z.B. der Fall, wenn der Ausschnitt Elemente besitzt, die mit Elementen aus anderen disjunkten Ausschnitten gepaart sind. Diesen Fall merken wir uns auch gleich in der Tabelle.

```
str = pairing.substring(start,start+len);
try
{
  NucleotideList nl = new NucleotideList();
  nl.setPairing(str);
} catch(IllegalArgumentException ex)
{
```

```
    zeroTable[start][len] = FAILURE_INDEX;
    return false;
  }
```

Unsere erfolgreiche Abbruchbedingung ist der triviale Fall, dass die Teilstruktur eine Länge von 1 besitzt und diese eine ungepaarte Base darstellt.

```
if (len == 1 && str.equals(":")) return true;
```

Danach gehen wir alle Generatoren des *gens* Attributes durch und rufen dabei ihre *apply()* Methode auf. Ist dies für einen Generator erfolgreich, so lässt sich die Teilstruktur mit Zuhilfenahme anderer kleinerer Teilstrukturen und diesem Generator erzeugen.

```
for (o=0;o<gens.length;o++)
{
  /* Only use 0-structure generators */
  if (gens[o].getAfterGapLength()==0)
  {
    if (gens[o].apply(start,len,this))
    {
      /* Generator fits, store its index (+1) */
      zeroTable[start][len] = ++o;
      return true;
    }
  }
}
```

Da wir alle Generatoren getestet haben, lässt sich auch die Struktur nicht erzeugen. Wir merken uns das in der Tabelle und geben dann *false* zurück.

```
zeroTable[start][len] = FAILURE_INDEX;
return false;
```

Analog zu dieser Methode ist *isCompositableOneStructure* implementiert. Diese erwartet die allerdings zwei Startpunkte und Längen als Parameter.

6.4.2 Die Klasse Generator

Die abstrakte Klasse Generator besitzt nur ein Attribut *active*, welches angibt, ob der Generator aktiv ist oder nicht. Mit *isActive()* und *setActive()* existieren Methoden, um dieses Attribut zu erhalten bzw. zu setzen. Es werden weiterhin verschiedene zu meist abtrakte Methoden definiert. Die interessantesten dabei sind die beiden *apply()* Methoden, jeweils einmal für Generatoren, die eine 0- und einmal für Generatoren, die eine 1-Struktur repräsentieren. Als Argumente erhalten sie die Startpunkte und Längen (*start* und *len* bzw. *foot1start*, *foot1len*, *foot2start* und *foot2len*) der Teilstrukturen und ein Composer Objekt (*comp*). Die Methoden überprüfen, ob dieser Generator auf die angegebene Teilstruktur angwandt werden kann und geben *true* zurück falls dem so ist, ansonsten *false*.

6.4.3 Die Klasse GeneratorLoop

Wie ihr Name schon sagt, implementiert diese Klasse den loop 0-Struktur-Generator. Betachten wir ihre *apply()* Methoden. Da der loop Generator eine 0-Struktur ist, ist nur diese Methode implementiert.

Wie weiter oben schon gesehen, müssen wir bei dem Loop Generator, die zu untersuchende Struktur in genau zwei nicht überlappende Teilstrukturen zerlegen, die zusammen wieder diese Struktur ergeben. Die erste Teilstruktur muss dabei eine 0-Struktur sein, die von der zweiten, einer 1-Struktur umschlossen wird. Wir iterieren also über zwei Variablen i und j, wobei i den Startindex der 0-Struktur und j den Startindex des zweiten Teils der 1-Struktur angibt. Dabei rufen wir die auf Strukturart testenden Methoden des *composer* Objekts auf. Die Implementation dieser Methode sieht wie folgt aus:

```
for (i=start;i<start+len;i++)
{
  for (j=i+1;j<start+len;j++)
  {
    if (composer.isCompositableOneStructure(start, i - start,
                                            j, start + len - j) &&
        composer.isCompositableZeroStructure(i, j - i))
    {
      return true;
    }
  }
}
return false;
```

6.4.4 Die Klasse GeneratorASCII

Dieser Generator implementiert alle Generatoren, für die eine Klammerndarstellung existiert. Hierfür übergibt man bei der Erstellung einer Instanz zusätzlich zum gewüschten Namen die Struktur des Generators als Zeichenkette. Zeichenketten, die nicht korrekt sind oder keinen Sinn machen (wie z.B. ein Generator bestehend aus einer ungebundenen Base) werden später ignoriert und was im *isValid* Attribut festgehalten wird. Neben dem Syntaxcheck mit der Hilfsklasse *NucleotideList* wird die Zeichenkette entsprechend repräsentierten Struktur in einer Datenstruktur als Attribut *nl* abgelegt. Die Anzahl der Elemente des Generators wird in *numbersOfElements* abgelegt. Für einen 1-Struktur Generator wird zusätzlich die Position der Lücke bestimmt und in einen privaten Attribut *gapPosition* gesichert.

Instanzen dieser Klasse fungieren als 0-Strukturen oder 1-Strukturen, daher sind beide *apply()* Methoden implementiert. Die Methode für 0-Struktur Generatoren ist jedoch einfacher in ihrer Implementation, deshalb werden wir nur auf diese hier eingehen. Das Grundprinzip, kann jedoch auf die 1-Struktur apply() Methode übertragen werden.

Da die Anzahl der Elememte eines solchen Generators nur zur Laufzeit bekannt ist, ist auch die Anzahl der notwendigen zu bildenen Teilstrukturen erst zur Laufzeit bekannt.

Daher wissen wir auch vorher nicht, wieviele Schleifen wir für die Unterteilung der Struktur benötigen, müssen alle möglichen Unterteilungen also in einer Schleife erzeugen.

Mit *loops* bezeichnen wir die gewünschte Anzahl der Schleifen. Dies entspricht der Anzahl der Elemente der Struktur abzüglich 1. Wir legen ein Feld i der Größe *loops* + 2 an. Dieses Feld repräsentiert die einzelnen Startpunkte der Teilstrukturen relativ zur Startposition der zu untersuchenden Struktur. Wir bezeichnen das Feld von nun an als *Konfiguration*. Den ersten Eintrag initialisieren wir sofort mit 0 und den letzen mit der Länge der zu untersuchenden Struktur (alle anderen werden durch Java ebenfalls mit 0 initialisiert).

```
loops = numberOfElements - 1;
i = new int[loops+2];
i[0] = 0;
i[loops+1] = len;
```

Als nächstes bestimmen wir die Anzahl der Schleifendurchläufe *maxk* für die Einzelschleife. Wir betachten zunächst die Elemente des Feldes i unabhängig von einander. Dann kann eine Startposition Werte zwischen 0 und *len* einnehmen. Da genau *loops* verschiedene Startposition variabel sind, existieren maximal len^{loops} verschiedene Konfigurationen.

```
maxk = 1;
for (j=0;j<loops;j++) maxk *= len;
```

Wir könnnen unsere Schleife nun einleiten. Es machen jedoch nicht alle Konfigurationen Sinn. Eine Konfiguration ist *gültig*, sofern ihre Elemente eine streng montone Zahlenfolge darstellen. Eine gültige Konfifuration repräsentiert also die oben besprochenene Intervalle. Für die weitere Bearbeitung, erlauben wir nur gültige Konfiguration. Wir halten dies in der lokalen Variable *isValid* fest.

```
for (k=0;k<maxk;k++)
{
  boolean isValid = true;
  for (j=0;j<=loops;j++)
  {
    if (i[j] >= i[j+1]) isValid = false;
  }

  if (isValid)
  {
```

Nun gehen wir jedes Element des Generators in einer inneren Schleife durch. Sie sind als Objekte der Klasse *Nucleotide* im Attribut *nl* abgelegt. Mit *getHead()* bekommen wir Zugriff auf das erste Strukturelement des Generators. Wir iterieren nun über j, welches als Index des aktuellen Elementes bzw. Teilstruktur interpretiert werden kann und bestimmen danach die absolute Startposition unserer ausgewählten Teilstruktur und der nachfolgenden Teilstruktur.

```
boolean success = true;
Nucleotide nucl = nl.getHead();

for (j=0;j<=loops;j++)
{
  int pos = start + i[j];
  int nextpos = start + i[j+1];
```

Ist das Strukturelement des Generators gepaart, so müssen wir überprüfen, ob die durch das Element und das gepaarte Element definierten Teilstrukturen, durch eine 1-Struktur Generator komponierbar sind. Wir bestimmen hierzu erst den Index des Partners (*pairIndex*) und daraus die absoluten Positionen der zugeordneten Teilstruktur bzw. der nachfolgenden Teilstruktur (*pairPos* und *pairNextPos*). Sollte *pairIndex* kleiner als j sein, so können wir die Schleife ohne Überprüfung fortsetzen, da wir dieses Paar schon in einer vergangenen Iteration überprüft haben.

```
  if (nucl.isPaired())
  {
    Nucleotide pair = nucl.getPairedNucleotide();
    int pairIndex = pair.index();

    /* check if we already analized this pair */
    if (pairIndex < j) continue;

    int pairPos = start + i[pairIndex];
    int pairNextPos = start + i[pairIndex+1];
```

Die Grenzen der Teilstrukturen sind jetzt alle bestimmt und wir können das *composer* Objekt benutzen, um zu erfahren, ob diese wirklich als eine 1-Struktur erzeugbar sind. Sollte dies nicht der Fall sein, so ist die zu untersuchende Teilstruktur nicht durch diesen Generator erzeugbar und wir brechen die Schleife ab.

```
    success = composer.isCompositableOneStructure(pos, nextpos-pos,
                                    pairPos, pairNextPos - pairPos);
    if (!success) break;
```

Analog gehen wir vor, wenn das Element ein ungepaartes Element ist. Jedoch ist der Fall um einiges leichter, da wir nur wir nur eine Teilstruktur benötigen, die eine erzeugbare 0-Struktur sein muss.

```
  } else /* !nucl.isPaired() */
  {
    success = composer.isCompositableZeroStructure(pos,nextpos-pos);
    if (!success) break;
  }
```

Wir gehen nun zum nächsten Element über. Ist die innere Schleife fertig und *success* gesetzt, heißt das, dass alle Teilstrukturen entsprechend der Generatorstruktur erzeugbar waren.

```
        nucl = nucl.getNext();
      } /* for (j=0;j<=loops;j++) */
      if (success) return true;
    } /* if (isValid) */
```

Die Konfiguration war nicht erfolgreich, daher wird eine weiter erzeugt. Die äußere Schleife endet, sofern wir alle möglichen Konfiguration erfolglos durchprobiert haben.

```
    /* Generate a new configuration */
    for (j=loops;j>0;j--)
    {
      i[j]++;
      if (i[j]<len) break;
      i[j]=0;
    }
  } /* for (k=0;k<maxk;k++) */
  return false;
```

6.5 Probleme

Die Sequenzen in der Pseudobase haben eine Größe von bis zu 140 Elementen, dabei sind nur die für Pseudoknoten relevanen Teilstrukturen enthalten. Die in der *Composer* Klasse benutzen Tabellen haben eine Speicherplatzbedarf von $O(n^2)$ und $O(n^4)$. In der vorgestellten Implementation beansprucht jeder Tabelleneintrag wenigstens ein Byte. Eine Tabelle für Einträge mit 140 Elementen würde demnach ungefär 366 Megabytes beanspruchen. Komplette RNA Sequenzen können mehrere tausende Nukleotide umfassen, daher gibt es noch Verbesserungsbedarf des verwendeten Algorithmus bei solch großen Sequenzen.

7 Auswertung

Dieser Abschnitt stellt die mit dem Programm gefundenen Ergebnisse dar. Ziel der Studienarbeit, war heraus zufinden, inwiefern sich mit den Generatorn in [1] biologische relevante Strukturen erzeugen lassen.

Wir haben bereits festgestellt, dass sich alle in der Natur vorkommende Strukturen erzeugen lassen, die keine Pseudoknoten beinhalten. Mit den ursprünglich definierten Generatoren können von 245 Strukturen mit Pseudoknoten, 49 erzeugt werden. Der Rest lässt sich nicht erzeugen, wobei anzumerken ist, dass 17 Stück zu groß waren (siehe Abschnitt 6.5) und somit nicht überprüft werden konnten.

Ursache hierfür sind Strukturen der Form `...(... : ...[)...:...]...`. Viele von diesen Strukturen lassen sich auf `[(])` verkleinern. Fügt man diesen Generator zu der

Generatoren	**Erzeugbar**	**Nicht erzeugbar**
concat, loop, lconcat, rconcat, stack =: *nested*	0	227
nested, inter	49	178
nested, inter, `([])`	221	6
nested, inter, `([])`, `( [)]`	221	6
nested, inter, `([])`, `( [)]`, `( :)`	227	0
nested, `([])`, `( [)]`, `( :)`	227	0
nested, `( [)]`, `( :)`	227	0
nested, `([) ]`, `(: )`	222	5
nested, `([) ]`, `( :)`	222	5
concat, loop, lconcat, stack, `( [)]`, `( :)`	227	0
concat, lconcat, stack, , `()`, `( [)]`, `( :)`	227	0
concat, lconcat, stack, , `()`, `([) ]`, `( :)`	227	0

Tabelle 4: Erzeugbarkeit von Strukturen in Abhängkeit von Generatoren.

Menge zu verwendender Geneartoren zu, so sind von den 228 überprüfbaren Strukturen, bereits 221 erzeugbar, dabei fällt auf, dass die Struktur der RNA mit Namen „CaYMV“ in der Datenbank fehlerhaft ist (es fehlen Paarungen).

Die Form der übrigen Strukturen lautet: `...(...[...)...(...]...)...`. Fügen wir zwei weitere Generatorn `( [)]` und `( :)` hinzu, so sind alle Strukturen, die getestet werden können, erzeugbar. Tabelle 4 gibt einen Überblick über diese und weitere Generatorkombinationen.

Literatur

[1] Michael Brinkmeier. *Pseudoknoted secondary RNA structure. 2003.*

[2] M. Zucker. *The using of dynamig programming algorithms in RNA secondary structure prediction. 1989.*

[3] R. Merkl, S. Waack. *Bioinformatik Interaktiv. 2002.*

[4] A. P. Gultyaev, F. H. van Bratenburg et. al *The computer simulation of RNA folding oathways using a genetic algorithm. 1995.*

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Studienarbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Ilmenau, den 9. Mai 2004